VI.05.

Немного попривыкли к интерфейсам, попробуем в этот раз рассмотреть/перечислить основные свойства интерфейсов в Go более формально. Завершим сегодня, ну, или почти завершим, разговор об интерфейсах в их техническом аспекте и потом пойдём дальше - к их применениям. Изложение материала в значительной степени следует статье *Martin Kock. A Guide to Interfaces in Go*, расположенной здесь : https://fast4ward.online/posts/a-guide-to-interfaces-in-go/.

И ещё одна полезная статья *Interface in Go (Golang)* лежит здесь: https://golangbyexample.com/interface-in-golang/.

Обе статьи в pdf (и линки на них) лежат в каталоге Materials.

Полиморфзм.

Интерфейсы дают нам полиморфизм - это уже мелькало, но попробуем сформулировать это чуть точнее. Пора, видимо, уже явно ввести и разъяснить смысл слова полиморфизм. Обратимся к Вики: *“Полиморфизм в языках программирования и теории типов — способность функции обрабатывать данные разных типов.”* ~~Будем говорить только о параметрическом полиморфизме (этих слов детям говорить ни к чему). Продолжая цитировать Вики, параметрический~~ *полиморфизм подразумевает исполнение одного и того же кода для всех допустимых типов аргументов*. Мы будем говорить только о таком виде полиморфизма (его ещё называют истинным, но тоже ни к чему множить термины). Ещё раз: интерфейсы дают нам возможность игнорировать тип переменной и вместо этого сосредоточиться на том, что мы можем с ней сделать (поведение, которое она предлагает нам с помощью своих методов), мы получаем возможность работать с любым типом, при условии, что он удовлетворяет интерфейсу.

Считается, что тип удовлетворяет интерфейсу, если и только если он реализует все методы, описанные этим интерфейсом - все методы интерфейса с заданным именем, причём все реализации методов имеют одинаковые наборы параметров и возвращаемых значений, как по количеству, так и по типам данных.

Заметим, что имена параметров и возвращаемых значений при описании типа неважны. Так, следующие два описания интерфейса VacationBenefeciar из примера sample.go

```
type VacationBenefeciar interface {
    SetVacation(duration time.Duration) VacationBenefeciar
    GetVacation(duration time.Duration) (v VacationBenefeciar, possible bool) // use
 vacation in whole or in part;
    // if the remainder is enough if the remainder is sufficient, then returns Vacat
ionBenefeciar with the modified
    // used part of vacation and true, else returns returns an unchanged value of Va
cationBenefeciar and false.
    VacationLeft() (remainder time.Duration) // vacation remainder.
}
```

и

```
type VacationBenefeciar interface {
    SetVacation(a time.Duration) VacationBenefeciar
    GetVacation(a time.Duration) (VacationBenefeciar, bool) // use vacation in whole
 or in part;
    // if the remainder is enough if the remainder is sufficient, then returns Vacat
ionBenefeciar with the modified
    // used part of vacation and true, else returns returns an unchanged value of Va
cationBenefeciar and false.
    VacationLeft() time.Duration // vacation remainder.
}
```

совершенно идентичны. В самом деле, снова посмотрим на пример: типы Permanent и Contract оба реализуют интерфейс VacationBenefeciar, но посмотрите, как написаны соответствующие методы:

```
func (p Permanent) SetVacation(duration int) VacationBenefeciar {
    p.vacation = duration
    return p
}

// get vacation in whole or partially of permanent worker
func (p Permanent) GetVacation(duration int) (v VacationBenefeciar, possible bool) {
    possible = p.vacation-p.used >= duration
    if possible {
        p.used += duration
    }
    v = p
```

```
    return
}

// vacation remainder of permanent worker
func (p Permanent) VacationLeft() (remainder int) {
    remainder = p.vacation - p.used
    return
}
```

что соответствует первому варианту, и

```
// set vacation duration of contract worker
func (c Contract) SetVacation(a int) VacationBenefeciar {
    c.vacation = a
    return c
}

// get vacation in whole or in part of contract worker
func (c Contract) GetVacation(a int) (VacationBenefeciar, bool) {
    if c.vacation < a+c.used {
        return c, false
    }
    c.used += a
    return c, true
}

// vacation remainder of contract worker
func (c Contract) VacationLeft() int {
    return c.vacation - c.used
}
```

в соответствии со вторым вариантом.

И хотя в коде интерфейс VacationBenefeciar записан, естественно, только одним - первым - способом, объявление типа Contract чудесно работает.

Тем не менее, понятно, что первый вариант объявления интерфейса сильно выигрывает в читабельностии, и записывать и интерфейсы, и методы, их реализующие, стОит так, как это сделано для Permanent, а не для Contract.

Интерфейсы в Go реализуются неявно.

Пара слов для тех, кто знаком хотя бы в общих чертах с объектами: для сравнения в классической объектной модели C++ полиморфизм требует, чтобы каждый тип наследовался от одного и того же базового класса; в Java этого не требуется, но в Java интерфейсы реализованы явно - это довольно жёсткое условие - то, что тип реализует некоторый интерфейс, следует явно указывать при объявлении этого типа.

В Go наследования нет вообще (хотя и об этом говорить не стОит), интерфейсы реализуются неявно. Что это означает? Так мы это уже знаем: как только тип реализует все методы интерфейса, он его сразу и реализует, этот интерфейс. В результате интерфейсы в Go позволяют реализовать полиморфизм чрезвычайно гибко - интерфейсы можно комбинировать, один тип может реализовывать несколько интерфейсов и т.д.

Ну, а раз интерфейсы реализуются неявно, то

Как узнать, что тип реализует интерфейс? Совместимость с интерфейсом.

И мы тоже уже знаем, как это делать - Go позволяет выполнить эту проверку. Пример такой проверки приводится в функции

```
func HasVacationLeft(s []Worker) (list []VacationBenefeciar) {
    for _, v := range s {
        if pv, ok := v.(VacationBenefeciar); ok {
            if pv.VacationLeft() > 0 {
                list = append(list, pv)
            }
        }
    }
    return list
}
```

Если такую проверку не делать (просто писать `pv := v.(VacationBenefeciar)` , то мы получим ошибку во время выполнения программы. Всё хорошо получается, но давайте посмотрим на ситуацию чуть глубже.

Преобразование интерфейса - это всегда проверка во время выполнения, которая может завершиться неудачно. И дело не в том, что проверка может завершиться неудачно, но если Freelancer используется только в такого рода проверках во время выполнения, то мы не можем знать наверняка, удовлетворяет ли Freelancer интерфейсу VacationBenefeciar. А часто бывает, что это хорошо бы знать, причём именно уже в процессе разработки

программы, чтобы видеть это уже на этапе компиляции, и это совсем не всегда просто так взять и увидеть - возможно, что интерфейс закопан где-то далеко в библиотеке. И сделать такую проверку в процессе разработки кода легко - просто пытаемся присвоить величину проверяемого типа переменной соответствующего типа, вот так:

```
var _ VacationBenefeciar = freelanc2
```

или ещё лучше даже так:

```
var _ VacationBenefeciar = Freelancer{}
```

В итоге получаем ошибку при компилировании. Компилятор всё заметил.

Реализация нескольких интерфейсов.

Да, один тип запросто реализует несколько интерфейсов. В примере `sample.go` все конкретные типы - Permanent, Contract, Freelancer, Pieceworker - реализуют интерфейсы Payee и Worker, а вот интерфейс VacationBenefeciar реализуется только типами Permanent и Contract.

Композиция интерфейсов. Встраивание (embedding) интерфейсов.

Go поддерживает создание именованного интерфейса из одного или нескольких других именованных интерфейсов. Таким образом мы можем повторно использовать небольшие интерфейсы для создания более крупных с помощью композиции. Это может быть особенно полезно, чтобы избежать ненужных преобразований во время выполнения для типов, которые имеют общее поведение. Да по-простому, если мы изменим вид метода в каком-то интерфейсе, то это метод автоматически изменится во всех интерфейсах, в которые это интерфейс встроен.

На самом деле это свойство Go-интерфейсов мы только что уже видели: интерфейс Worker включает в себя интерфейс `Payee`. В принципе, можно было бы построить интерфейс

```
type PrivilegedWorker interface {
    Worker
    VacationBenefeciar
}
```

и мы это сделали.

Или, например, когда мы строили бинарное дерево поиска и АВЛ-дерево, у нас появился интерфейс `Key`, полученный в результате композиции интерфейсов `Ordered` и `Visual`:

```
type Ordered interface {
    Before(b Ordered) bool
}

type Visual interface {
    Show() string
}

type Key interface {
    Ordered
    Visual
}
```

Скажем явно: при объявлении интерфейса в него можно включать не только методы, но и другие интерфейсы. Зачем, мы же, вроде, можем просто тупо повторить при объявлении интерфейса все функции из включаемого в него другого интерфейса. Ну, всё-таки это не совсем одно и то же, хотя бы с точки зрения формы, внешнего вида объявления интерфейса. И это весьма существенно - мы явно видим, что какой-то интерфейс является “укрупнением” другого. Так может и не нужно “укрупнять” интерфейсы? Тоже совсем не так: новый интерфейс - новый тип, к нему можно ассертить и проверять возможность этого действия, его можно использовать в объявлениях функций и методов и т.д. Скажем так:

Интерфейсы надо измельчать. Но не в пыль.

Интерфейсы в Go обязательно должны быть не больше, чем нам нужно в любой конкретной ситуации. И для каждой возможной ситуации требуется свой набор методов, свой интерфейс. Иначе говоря, интерфейс должен включать все необходимые методы и никаких больше.

В примере `sample.go` не все работники имеют право на оплачиваемый отпуск, т.е. интерфейс `VacationBenefeciar` нельзя включать в интерфейс `Worker`. Хотя вполне себе возможны функции, работающие именно с такими работниками, и тогда есть смысл завести интерфейс `PrivilegedWorker`, о котором уже говорилось чуть выше. Он, собственно, есть в примере `sample.go`, хотя и не используется.

Значение интерфейса (переменной интерфейсного типа).

И об этом мы уже упоминали, но давайте сформулируем всё вместе, раз уж такой разговор. Любая переменная интерфейсного типа в каждый момент имеет свой базовый (он же конкретный) тип. Интерфейс - это набор методов, которые он предоставляет, Но для того, чтобы программа выполняла метод, для вызова метода переменная интерфейсного типа должна обеспечивать доступ к фактическому значению величины. Так, например при выполнении функции `func totalExpense(s []Worker) int` мы работаем с `Worker` ами, но каждый раз в цикле фактическое значение этого `Worker` а имеет свой конкретный тип. И мы уже говорили, что переменная интерфейсного типа хранит как фактическое значение, так и конкретный тип этого фактического значения. Это хорошо видно в коде:

```
    for _, v := range employees {
        fmt.Printf("Type = %T. Value = %v.\n", v, v)
    }
```

выводящем

```
Type = main.Permanent. Value = {1001 2500 20 21 3}.
Type = main.Permanent. Value = {1002 3000 30 15 15}.
Type = main.Contract. Value = {2002 2400 10 10}.
Type = main.Freelancer. Value = {4001 30 120}.
Type = main.Freelancer. Value = {4003 45 80}.
Type = main.Pieceworker. Value = {5002 [450 250 430 700 315]}.
```

И сразу здесь чудесное

Упражнение.

1. Залезть внутрь переменной интерфейсного типа и посмотреть, как она устроена, и что там лежит по тем адресам. Что-то подобное тому, что мы делали в 5-м семестре, залезая внутрь массивам, слайсам, строкам, структурам, используя package unsafe.
2. Посмотреть, что происходит при переходе величины ,базового (конкретного) типа к интерфейсному. Ну, т.е. сравнить, что находится внутри переменных perm1, p, w, b и pw:

   ```
   var p Payee = perm1
   var w Worker = perm1
   var b VacationBenefeciar = perm1
   ```

```
var pw PrivilegedWorker = perm1
```

Пустое (нулевое) значение интерфейса (переменной интерфейсного типа).

И вот тут мы сталкиваемся с тем, что надо чётко разделять неинициализированное значение переменной интерфейсного типа (которое есть nil) и инициализированную переменную интерфейсного типа, которая имеет nil в качестве фактического значения (например, фактическое значение есть пойнтер или неинициализированный слайс).

Всё это иллюстрирует следующий фрагмент кода:

```
var w Worker
fmt.Println(w)
//    <nil>
fmt.Printf("w is %v and has value of type %T", w, w)
//    w is <nil> and has value of type <nil>panic: runtime error:
//    invalid memory address or nil pointer dereference
fmt.Println(w.ID())
/*
    [signal 0xc0000005 code=0x0 addr=0x0 pc=0x9e699]

    goroutine 1 [running]:
    main.main()
            sample.go:212 +0x619
    exit status 2
*/
```

Разумная стратегия на такие случаи - просто не допускать того, чтобы фактические значения интерфейсных величин когда бы то ни было становились nil'ом, ну, или чтобы методы этого интерфейса корректно обрабатывали nil-значения.

Пустой (empty) интерфейс.

Мы можем определить пустой интерфейс - интерфейс без каких-либо методов на нем:

```
type emptyInterface interface {}
```

Пустой интерфейс не имеет никаких требований, и поэтому мы можем присвоить всё, что

угодно, переменной или параметру этого типа. Хорошо это или плохо - однозначного ответа нет, но стОит обходиться с пустым интерфейсом очень осторожно. Пустые интерфейсы используются при работе с несколькими различными типами, которые не имеют ничего общего, т.е. не имеют общих методов, при работе с потенциально непредсказуемыми типами, как, например, с двусвязным списком (doubly linked list) из `package container/list`.

В чём же опасность применения пустого интерфейса? Неуместное использование пустого интерфейса ведёт к потере важнейшей характеристики кода - безопасности типа (type safety). Пустой нтерфейс ничего не говорит о своих свойствах. Используя пустой интерфейс, мы не знаем, что у нас есть, мы не можем делать никаких предположений об этом.

Конечно, если мы всегда знаем, что мы передаем, то мы можем получить величину (например, принять её как аргумент) и принудительно преобразовать её тип. Но, если мы пишем что-то библиотечное, то что мы знаем о применении этой библиотеки - да ничего. Да и вообще, и это главный побудительный мотив, программа - живой организм, программы развиваются, меняются, и то, из чего мы исходим сегодня, завтра может измениться, сегодня всё работает - завтра всё рушится. Пустой интерфейс ничего не говорит о себе, и его следует применять исключительно в тех случаях, когда мы действительно ничего не можем предполагать о получаемых для обработки данных.

Даже когда мы взаимодействуем со сторонним API, у нас будут какие-то ожидания в отношении ответа, и, например, если мы получаем данные в JSON, то при разворачивании этих данных в какой-то собственный тип у нас есть обоснованные предположения о каких-то ождаемых полях.

Говоря про измельчение интерфейсов, прозвучал тезис: “интерфейс должен включать все необходимые методы и никаких больше”. Говоря о пустом интерфейсе, очень уместно повторить эту фразу, только акцентируясь не на второй части, а на первой - все методы. Да, надо передавать ВСЁ, что мы предполагаем о величине, передаваемой на обработку (и ничего больше, конечно).

Так что пустые интерфейсы не совсем безопасны. При работе с пустыми интерфейсами следует не делать никаких предположений о них и проверять их на каждом шагу. Разумеется, пустые интерфейсы могут быть преобразованы в пользовательские типы и именованные интерфейсы, как и обычные интерфейсы.

Вот (дурацкий, надо сказать) пример. Рассмотрим функцию

```
func PrintHasVacation(w interface{}) {
```

```
    if _, ok := w.(Worker); ok {
        _, ok = w.(VacationBenefeciar)
        fmt.Println(ok)
    }
}
```

принимающую пустой интерфейс. И, поскольку пустой интерфейс реализуется любым типом, то мы можем передать в эту функцию всё, что угодно. Рассмотрим вот такой вызов

```
    PrintHasVacation("Vasja Pupkin")    //    ignore
```

Аргумент “Vasja Pupkin” не относится к тому типу, который мы ожидаем. Варианты реализации такого сценария: вернуть ошибку или проигнорировать её, в зависимости от ситуации. В нашем примере выбран второй вариант.

Да, мы внутри функции всё проверяем, и проблемы разрешаются. Но предствим себе, что по ходу развития программы появится какой-то новый тип, который мы передаём как пустой интерфейс. Нам придётся лезть в то место, где обрабатывается его вызов. А это и громоздко, и довольно часто просто невозможно - код находится где-то вне нашего доступа.

Ну, и ещё один вопрос. Он не видится сейчас таким уж важным, скорее даже лучше его обйти, но пусть будет для полноты картины. Да и вполне себе может быть, что поговорить об этом будет уместно уже в это раз. Но, в общем-то, это опционально.

Частично экспортируемые интерфейсы

На неэкспортируемый интерфейс (имя интерфейса начинается с маленькой буквы), естественно, можно ссылаться только из его собственного пакета. На экспортируемые интерфейсы можно ссылаться из других пакетов, но как насчет экспортируемых интерфейсов, которые содержат как экспортируемые, так и неэкспортируемые методы? Возьмем интерфейс School в качестве примера:

```
// School is defined in package school
type School interface {
    Name() string
    Location() string
    students() []string
}
```

```
// University is defined in other package schools
type University struct {}

func (u University) Name() string { return "Copenhagen University" }
func (u University) Location() string { return "Copenhagen City Center" }
func (u University) students() []string { return []string{"George", "Ben", "Louise",
 "Calvin"} }
```

Проверим совместимость интерфейсов, как было предложено выше:

```
var _ school.School = University{}
```

и получим сообщение об ошибке (на этапе компиляции):

```
// cannot use University literal (type University) as type school.School in assignme
nt:
//    University does not implement school.School (missing school.students method)
//                have students() []string
//                want school.students() []string
```

Тип University пытается реализовать интерфейс school.School, но не может этого сделать, поскольку не может реализовать метод students интерфейса school.School - к нему просто нет доступа, пакет school просто не видит этой попытки реализации.

Это означает, что из-за неэкспортируемого метода мы можем реализовать интерфейс School только в пакете school, но не в других пакетах.

Как же можно использовать частично экспортируемый интерфейс? Давайте попробуем встроить интерфейс School:

```
// University is defined in other package
type University struct {
    school.School
}

func (u University) Name() string { return "Copenhagen Highschool" }
func (u University) Location() string { return "Copenhagen City Center" }
```

Проверяем:

```
var _ school.School = University{}
```

И это работает. Работает до тех пор, пока мы не пытаемся переопределить метод students.

Таким образом, если мы пишем пакет, в котором определяем интерфейс, содержащий неэкспортируемые (несообщаемые) методы, мы можем гарантировать, что любой тип, который хочет реализовать этот интерфейс, не переопределяет наши неэкспортируемые методы. По сути, мы заставляем реализации внедрять типы, экспортируемые нашим собственным пакетом, причём любой вызов неэкспортируемого метода всегда будет обрабатываться внутри, реализацией этого нашего собственного пакета.

И, соответственно, реализации не могут вызывать несообщенные методы. Только ваш собственный код может это сделать. Таким образом, частично экспортированные интерфейсы в основном являются механизмом обеспечения того, чтобы ваш собственный пакет предоставлял некоторые полезные методы для внутреннего использования. Внешние пользователи вашего пакета могут ссылаться на интерфейс, но не могут встроить его в свои собственные типы.

Заключение

Интерфейсы в Go являются неявными и структурно типизированными. Как мы уже видели, они чрезвычайно универсальны. Пустой интерфейс соответствует чему угодно, и непустые интерфейсы могут быть преобразованы как в определенные типы, так и в именованные интерфейсы, а также в анонимные интерфейсы и специальные интерфейсы с другой структурой.

Значения интерфейса имеют базовый тип. Они могут быть составлены и использованы без полной реализации их методов. Они также могут быть реализованы несколькими различными типами, каждый из которых реализует только часть интерфейса, но при композиции реализуют его полностью.

И важно понять, что Go применяет интерфейсы немного иначе, чем в других языках, в других моделях реализации полиморфизма. Но об этом - в ближайшее время.